

AD-A197 197

Unclassified

DTIC FILE COPY

4

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Parallel Debugging Using Graphical Views		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) M. Bailey, D. Socha, D. Notkin		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0264
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE March 1988
		13. NUMBER OF PAGES 9
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) <div style="border: 1px solid black; padding: 5px; text-align: center;">DISTRIBUTION STATEMENT A Approved for public release; Distribution Unlimited</div>		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) debugging, parallel programming program visualization, parallel debugging monitoring		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Graphical views are essential for debugging parallel programs because of the large quantity of state information contained in parallel programs. Voyeur, a prototype system for creating graphical views of parallel programs, provides a cost-effective way to construct such views for any parallel programming system. We illustrate Voyeur by discussing four views created for debugging Poker programs. One is a general trace facility for any Poker program. The other three are tailored to display a specific type of algorithmic information. Each of these views has been instrumental in detecting bugs that would have		

DD FORM 1473

JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

have been difficult to detect otherwise, yet were obvious with the views.



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By <i>per HP</i>	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
<i>A-1</i>	

## Parallel Debugging Using Graphical Views

Mary Bailey, David Socha, and David Notkin  
Department of Computer Science, FR-35  
University of Washington  
Seattle, Washington 98195

TR 88-03-05  
March 1988

Graphical views are essential for debugging parallel programs because of the large quantity of state information contained in parallel programs. Voyeur, a prototype system for creating graphical views of parallel programs, provides a cost-effective way to construct such views for any parallel programming system. We illustrate Voyeur by discussing four views created for debugging Poker programs. One is a general trace facility for any Poker program. The other three are tailored to display a specific type of algorithmic information. Each of these views has been instrumental in detecting bugs that would have been difficult to detect otherwise, yet were obvious with the views.

This research funded in part by the Office of Naval Research Contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878, and the Air Force Office of Scientific Research Contract 88-0023.

## Parallel Debugging Using Graphical Views<sup>1</sup>

*Mary L. Bailey, David Socha, and David Notkin*  
Department of Computer Science, FR-35  
University of Washington  
Seattle, WA 98195

### Abstract

Graphical views are essential for debugging parallel programs because of the large quantity of state information contained in parallel programs. Voyeur, a prototype system for creating graphical views of parallel programs, provides a cost-effective way to construct such views for any parallel programming system. We illustrate Voyeur by discussing four views created for debugging Poker programs. One is a general trace facility for any Poker program. The other three are tailored to display a specific type of algorithmic information. Each of these views has been instrumental in detecting bugs that would have been difficult to detect otherwise, yet were obvious with the views.

---

<sup>1</sup>This research funded in part by Office of Naval Research Contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878, and Air Force Office of Scientific Research Contract 88-0023.

# Parallel Debugging Using Graphical Views<sup>1</sup>

Mary L. Bailey, David Socha, and David Notkin  
Department of Computer Science, FR-35  
University of Washington  
Seattle, WA 98195

## 1. Introduction

Graphical views are essential for debugging parallel programs because of the large quantity of state information contained in parallel programs. Voyeur, a prototype system for creating graphical views of parallel programs, provides a cost-effective way to construct such views.

Historically, computers have supported debugging by providing access to the program's state. The programmer assimilates this information, comparing the expected and actual state of computation to validate the program's execution or detect errors. Today's workstations allow a more powerful approach to debugging. Graphical views can synthesize images of the program's state, focusing the image on the algorithmic structure of the program or on the architectural structure of the target computer. These images present a great deal of information in a readily assimilated manner. They are a great help in managing state information, especially for parallel programs with their orders of magnitude more information than sequential programs.

Current parallel debugging tools provide one of three levels of debugging. One is to use a sequential debugger on one of the processes of a parallel program [1]. A second is to provide a textual trace of program execution [2]. A third is to integrate the trace information and a visual view of the program. Belvedere [3] displays a graph of a message passing program and shows the message activity on the edges of this graph. Belvedere also can find and display logical patterns of message activity in an asynchronous message-passing program. SDEF [4] uses a display similar to the Poker [5] Trace View described below. In addition they show message activity.

Voyeur is a prototype system for constructing general graphical views of parallel programs. The goal in developing Voyeur is to make creating new views for debugging specific algorithms practical. While the technology has been designed for viewing Poker programs, it is not limited to Poker and can easily be made to work for any sequential or parallel program. We have used Voyeur to construct four views of Poker parallel programs, each providing a different form of debugging information. One of these views is a general debugging tool for Poker programs; the other three were developed for specific algorithms. The last view took only three days to complete, and we expect development time to decrease as more general support is added to the system. These views have been instrumental in finding bugs in Poker programs, bugs that would have been difficult to detect otherwise.

This paper first presents a brief overview of the Poker and its debugging facility Trace View. Then we describe the Voyeur views, and discuss their role in debugging Poker programs. Next we briefly discuss Voyeur's structure and the process of creating new views. Finally, we present our conclusions and future directions.

## 2. The Poker Programming Environment

Poker is a programming language for defining and executing non-shared memory MIMD parallel algorithms. Poker programmers use a special graphical programming environment, which requires that five separate development views be defined. Using the Poker programming environment, the programmer: (1) draws connections between processing elements (PEs) to define the communication graph (such as a tree or mesh); (2) defines the sequential programs (written in Poker C) using standard editing and compilation tools; (3) graphically associates the sequential programs with specific PEs (for instance, assigning different programs to the root and leaves of a tree); (4) associates symbolic names with each communication port on a PE, increasing the flexibility of sequential programs that run at each PE; and (5) assigns files to I/O pads, connecting the program to an external file system.

---

<sup>1</sup>This research funded in part by Office of Naval Research Contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878, and Air Force Office of Scientific Research Contract 88-0023.

## 2.1. Poker Trace View

Poker programs can be run on several parallel architectures[6], such as the CHiP[7] and the Cosmic Cube[8]. But during development and debugging, most programs are executed using a simulator that runs on sequential machines. The environment contains a run-time view, the Trace View (see Figure 1), that integrates the simulator into the environment.

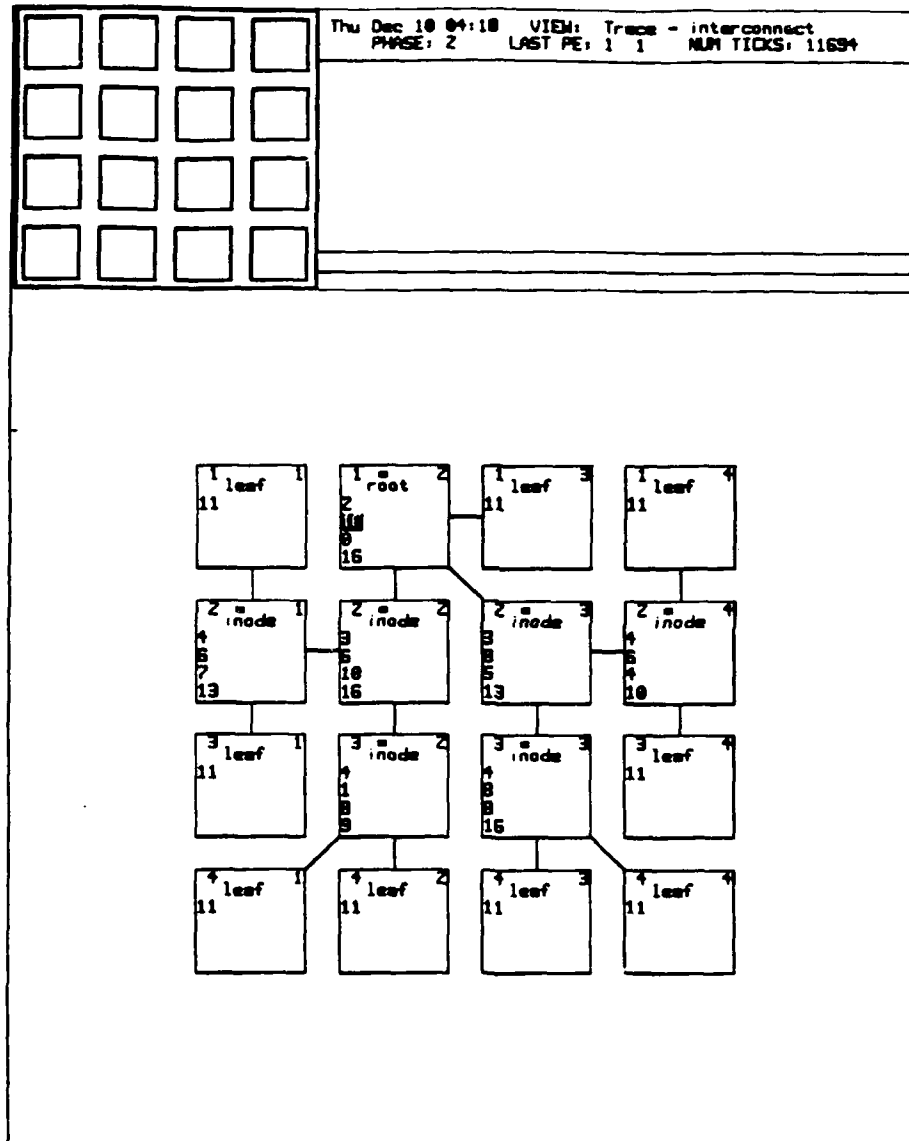


Figure 1. Poker's Trace View.

The Trace View presents the PEs arranged as their communication graph, including what code they are executing (root, inode, leaf), the PE (i,j) index in the PE grid, and the state of execution ('=' means running). Before compilation, the programmer inserts trace statements in the PE codes indicating variables to trace. The compiler then automatically generates code to send new values of the variables back to the Trace View whenever the variables changes values. Poker's Trace View displays up to four of trace variable values in each of the PE windows. The latest values are highlighted.

Above the display of the PEs in Figure 1 are two areas. On the left is a representation of the entire PE grid. When the screen is too small to fit all of the PEs on the grid, this section will have a box that outlines the area of PEs that are visible on the screen below it. On the right is an area displaying status information and messages, as well as a command line used both to control the execution of the Poker program and also to access and modify the state of the program.

This view has several useful features. The visual structuring of the trace information according to the program structure greatly helps in seeing patterns during program execution. Highlighting recent values pinpoints the interesting data among the wealth of data shown. The execution status quickly informs the user when a PE has stopped and whether it encountered a run-time error. Finally, the user has a great deal of control over program execution, being able to single step or execute at full speed, all while watching the changing state of computation.

## 2.2. Voyeur Poker Trace View

The Voyeur Poker Trace View is an enhanced version of Poker's Trace View. It is aimed at fixing two problems of Poker's Trace View:

1. The view's structure and power is inflexible. The display is tied to the program's algorithmic description, with the PEs in their grid arrangement. Each PE contains a maximum of four lines of alphanumeric data. There is no facility for tracing or displaying dynamically allocated structures such as linked lists. The traceable types of data are fixed by the compiler.
2. The view is fixed in stone. The design of Poker's environment does not admit ready extension or modification of the view.

Figure 2 shows Voyeur's version of a Poker Trace View. The status information is similar to Poker's. The command line, on the other hand, has been replaced by a set of buttons for frequently used commands, and menu items for setting parameters or modifying the view's format. Messages from the view to the user are displayed in the status area or bundled in notifiers if an immediate response is needed. Using buttons and menus gives a more usable interface.

The most important enhancement is an increased flexibility in the display of the PE area. The trace variable area contains the name of each variable as well as its value. There is no restriction on the number of variables shown for each PE. The PEs boxes, while still arranged in their grid reflecting the communication structure of the algorithm, can be re-sized to show more (or less) information, and individual PEs windows may be copied, moved, and re-sized independently of the PEs in the grid. The scroll bars move the PE grid within the PE grid window.

Even with the large screens on today's workstations, complex programs with more than a dozen trace variables per PE easily exceed the screen space. The flexibility of Voyeur's PE window sizes and duplicate PE windows allows the user to view a small amount of information for all PEs, for detecting global patterns, while still seeing complete information for a few PEs. Alternatively, the grid PEs can show complete information while the duplicate PEs show PEs that have been pushed off the edge of the screen.

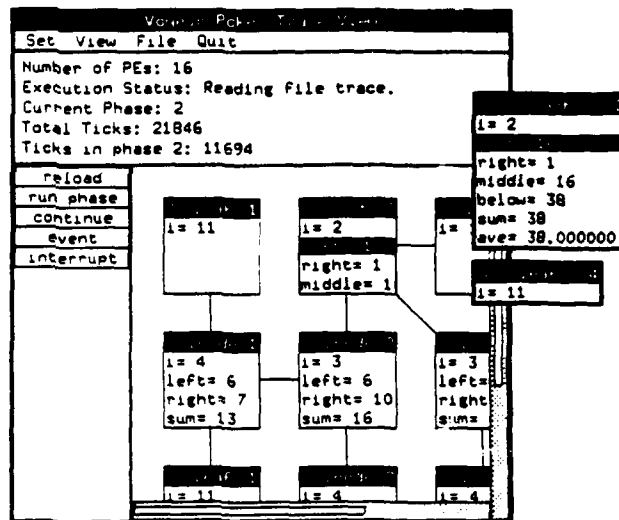


Figure 2. Voyeur Poker Trace View.

### 3. Other Voyeur Views

Voyeur's Poker Trace View provides an improved version of Poker's Trace View. Other views provide assistance in finding other types of bugs. This section describes three other views and how they helped find bugs in Poker programs. The variety and complexity of some of the bugs is interesting in itself. We will illustrate the use of views by giving examples of bugs found in a Poker program being used to investigate ways to dynamically balance the work load for non-shared memory algorithms.

The algorithm simulates sharks and fishes moving in a two-dimensional grid of (x,y) points (see Figure 3) [9]. Sharks and fishes inhabit points, one animal per point. Each animal may move one unit up, down, right, or left, but not off the edge of the grid. Fish are further constrained to move only into vacant points. Sharks may not move to a point containing a shark but they can, and prefer to, move to an adjacent point containing a fish and thus eat it. If there are a selection of available points, the shark or fish randomly chooses one. Both species occasionally give birth, with the baby staying in the place the parent vacates. Sharks starve if they have not eaten in a while. There is an infinite supply of plankton, so fish never starve. To simplify the algorithm, evolution alternates between moving all of the fish and all of the sharks.

To simplify the allocation of the grid space to PEs, we divide the world into slices, where each slice is a column in the grid. These slices are allocated to the PEs, which are connected in a single line, such that the order of the slices in each PE as we traverse from left to right is the same as in the grid. This order must be maintained. Slices within a single PE are chained along a linked list. Within each of these slices, active data points are chained in an orthogonal linked list, indicated by the variable sized columns under the grid.

Periodically, the algorithm checks to see if there is an imbalance in the current allocation, i.e., if some PEs have much more data (and hence more work to do) than other PEs. If so, re-balancing occurs and the slices are moved to balance the data. Figure 3 shows the effect of re-balancing for this example. Before re-balance the PEs have 4, 2, 0, and 6 data points, respectively. After re-balancing, they have 3, 3, 4, and 2 data points, respectively. Note that slices are atomic; all data points within a slice must be contained in a single PE, so there may not be equal numbers of data points in the PEs after re-balancing.

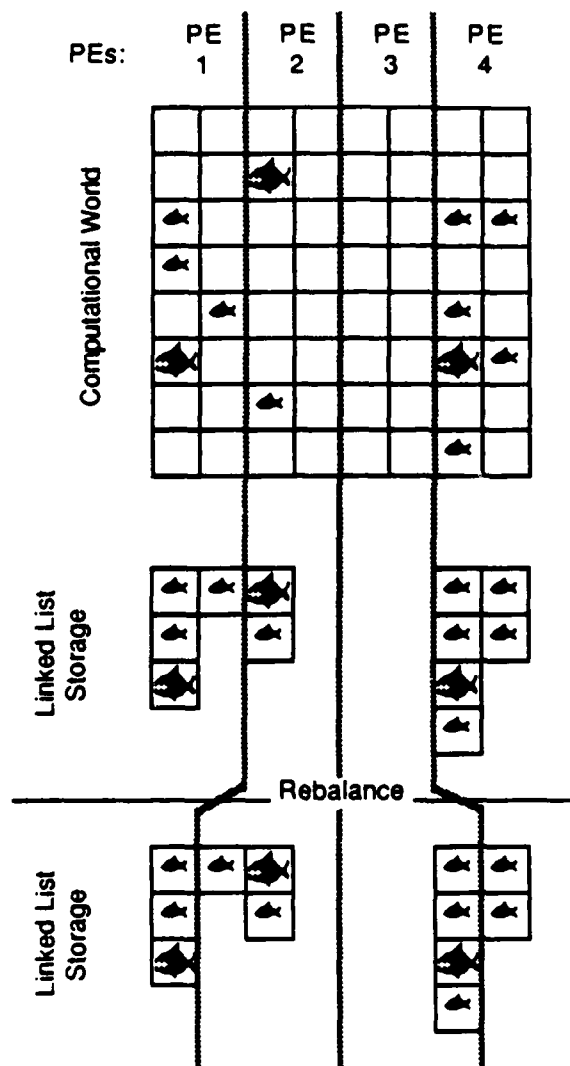


Figure 3. Sharks & Fishes algorithm.

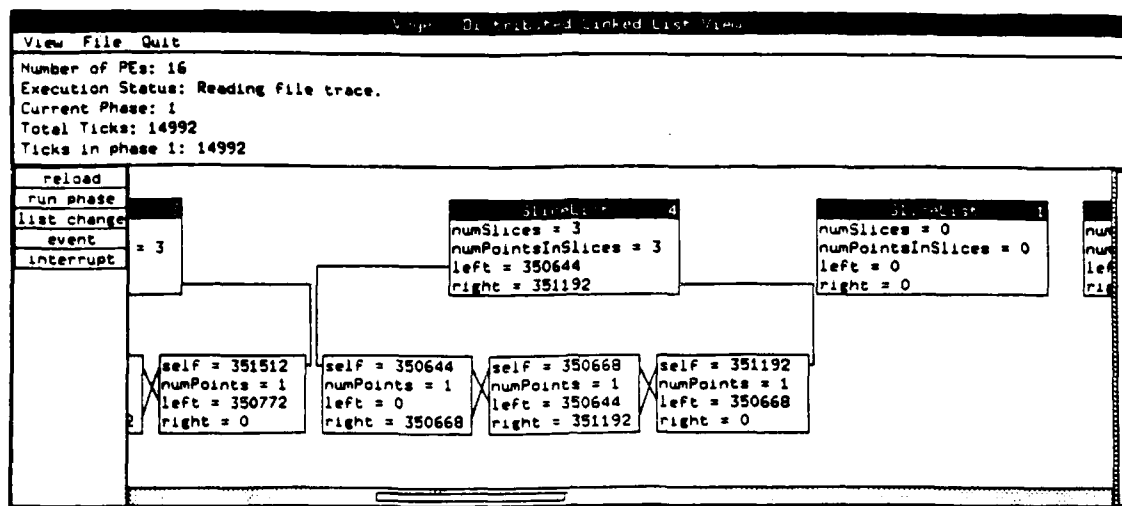


Figure 4. Voyeur's Distributed Linked List View.

### 3.1. Distributed Linked List View

Even the more flexible interface of the Voyeur Trace View provides little help when tracing dynamic structures like linked lists. Dynamic structures have no fixed size and are most easily viewed in terms of their structure. For this reason, we created a Linked List View to show the state of the linked list used to store the sharks & fishes grid (see Figure 4).

The status and control information of the Distributed Linked List View is similar to that in the Voyeur Trace View described above. The difference is that the PE grid has been replaced by a linked list view. At the top of the linked list is a row of header cells, one for each PE, indicating the number of slices in that PE, the total number of data points in the slices, and the values of the pointers to the left-most and right-most slices in the PE. Centered below each header cell is a linked list of slice headers, each indicating how many data points are occupied in the slice and the values of the left, right, and self pointers.

The first use of this view detected a bug in the linked list portion of sharks & fishes. When we deleted a slice from a PE's linked list we forgot to update the pointers to that no-longer-existing slice. This bug was immediately visible in the linked list view, yet had gone undetected during debugging with the Voyeur Trace View. The Distributed Linked List View also allowed us to easily verify that the slices were moving correctly from one PE to the another during the re-balance phase of the program.

This view illustrates one of the philosophies underlying Voyeur: present the state of the information within the program, as well as synthesized information. The connecting edges do *not* contain the same information as the pointer values, since an incorrectly written Poker program could enter a state that was unexpected by and inconsistent with the Voyeur view. Thus, the Distributed Linked List View shows the actual value of the left, right, and self pointers as well as the synthesized links connecting slices in order to aid the programmer in detecting expected *and* unexpected types of errors.

### 3.2. Icon View

To determine if the sharks and fishes were moving correctly within and among PEs we developed a view that shows icons of objects located on a (x,y) coordinate grid (see Figure 5). Again, this view has a few menus, a small status area, and a set of control buttons for commands. The display area shows the icons present at each location in the grid, and, optionally, the allocation of the grid points to PEs. The generation button consumes the next snapshot of the grid as generated by the simulator.

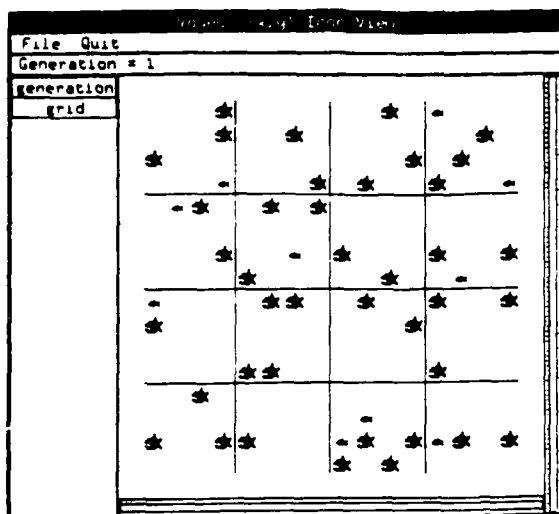


Figure 5. Voyeur's (x,y) Icon View.

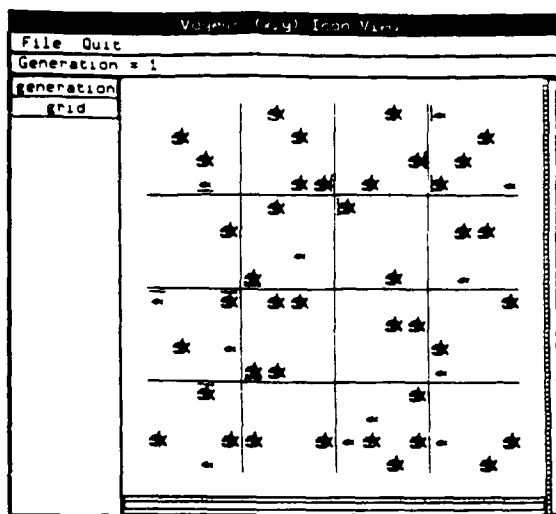


Figure 6. Using Voyeur's (x,y) Icon View to show knowledge of adjacent PEs.

This view has been used to find some rather subtle and instructive bugs. Near the beginning of programming sharks & fishes, before the Icon View was available, we programmed the fish to randomly choose among the vacant adjacent spots. We laboriously used a standard sequential debugger (dbx [10]) to follow the execution of one of the PEs in our light-weight process simulator. The two fish we traced within the PE randomly moved west, which is reasonable. However, when we viewed the program from Voyeur's Icon View, we saw *every* fish in *every* PE "randomly" move west. Clearly, there was a problem with our use of random numbers, which we quickly traced to an incorrect setting of a variable.

This bug is interesting since it could have been very difficult to detect locally, yet it jumped out at us when we saw the global behavior. Furthermore, the view extracted the essence of the algorithm - where the fish were and where they moved to -- without requiring us to wade through the actual instructions used to move the fish.

A second bug reaffirmed the value of global information. At this point, we were allocating the grid points to the PEs by dividing the grid into a set of 4x4 squares and connecting the PEs in a mesh, as reflected in the grid lines in Figure 5. When watching the fish move, we noticed that some fish on the east side of a PE jumped, in one move, across to the west side of the same PE. In one place in the program, we had reversed the constants EAST and WEST. Not only was this bug obvious when watching the view, but noticing that fish never jumped between the north and south edges supplied more data for finding the bug.

A third interesting bug occurred when we started moving fish across PE boundaries. The first clue that something was wrong was that sharks adjacent to a fish in bordering PE were not eating the adjacent fish. After getting more information from the simulator about what each PE thought was in adjacent PEs, and creating icons to indicate what each PE thought was in the border of adjacent PEs (| for a shark to the left, | for a fish, and so on), we discovered that PEs in the lower right section were not correctly transmitting their edge information (see Figure 6). We had replaced instances of a constant, 4, describing the width of a PE's area with a variable and changed the local coordinate system from being relative to the upper left corner of each PE to being relative to the upper left corner of the entire grid. However, we had missed a few constants so that points with coordinates greater than 4 were being ignored when passing information to adjacent PEs.

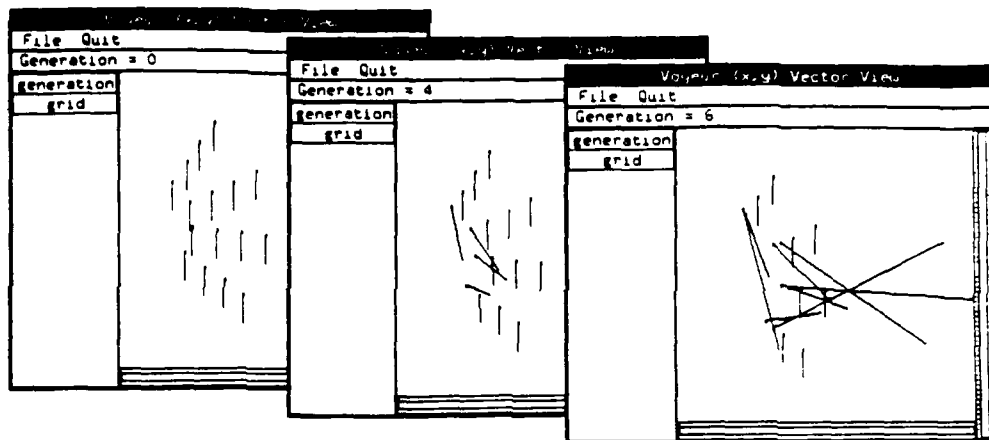


Figure 7. Voyeur's (x,y) Vector View.

### 3.3. Vector View

Our fourth view plots vectors on an (x,y) grid. The Vector View (see Figure 7) has the same status, menus, and control buttons as the Voyeur (x,y) Icon View. The difference is that it plots vectors with squares at the origin.

We developed this view from the (x,y) Icon View by adding a setup file for increased flexibility. Each line of the file specifies the type of an object to view (icon or vector), the simulator's unique identifier for that object, and the file containing the bitmap for that object. Adding more objects, changing an object's appearance, or removing an object is as easy as modifying this file.

This view was used by a colleague in the Applied Math department to see if the explosion of the SIMPLE [11,12] calculation running under Poker was due to an algorithmic problem or a numerical instability resulting from the sparseness of points in the 3-dimensional space. By viewing the evolution of the vectors across time he could see the vectors crossing just before the simulation blew up, indicating numerical instability instead of an algorithmic error (see Figure 7).

## 4. Integrating Voyeur and Programs

*While having the various views has been invaluable in debugging Poker programs, the key to Voyeur's usefulness is the ease of creating new views. This is facilitated by Voyeur's structure. The structure of the Voyeur prototype is shown in Figure 8. Boxes with square corners are heavy-weight processes. Boxes with round corners are modules. Messages from the user filter down to change the form of the view or to request more simulation data. Messages from the simulator filter up to change the state shown by the view.*

A Voyeur view consists of the simulator interface, the adapter, the modeler, and the renderer [13]. The adapter translates between the string-based simulator messages and the procedural interface of the modeler and renderer. These messages may come directly from the simulator, or may come from a trace file produced by the simulator. Based on the type of each simulator message, the corresponding procedure for that message is called. The modeler manages data specific to the application. The renderer defines the user interface (based on X-windows [14]), which is responsible for drawing the view of the modeler, for manipulating the form of the view, and for letting the user control the program's execution. Just as control events from the X-window interface drive the execution of the renderer and modeler, state messages from the simulator drive the adapter, the modeler, and the renderer.

The user interfaces of the views share a basic structure. The view's title is contained in a title bar at the top of the view. Underneath the title bar is a set of pull-down menus. These can be fairly complex, as in the Trace View, or can simply contain a facility for quitting, as in the (x,y) Icon View. Below the menu bar is a status area containing data appropriate for the view. Below this and

to the left is a set of control buttons for controlling the execution of the simulation. The data area is in the lower right-hand corner, and contains scroll bars for movement within this area.

To create a new view, the user first annotates the Poker program to send appropriate messages to the view. The adapter is automatically created from a description of these messages. The user must then write the modeler and the renderer. Because much of the user interface is common to all views, creating the renderer consists of modifying an existing one. For example, creating the Distributed Linked List View from the Trace View took only three days.

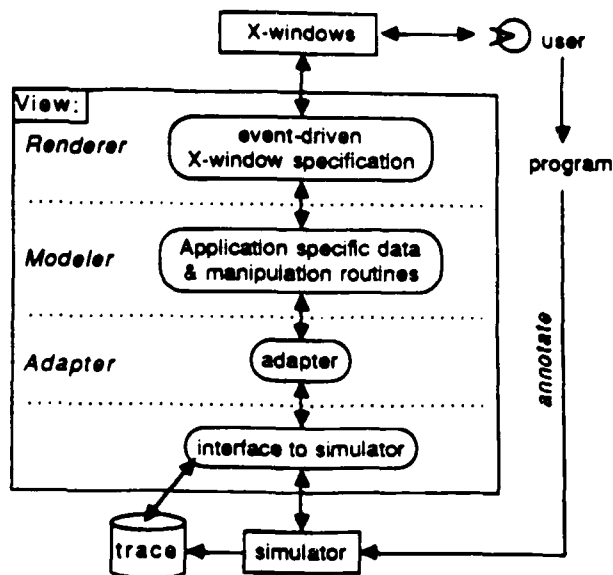


Figure 8. Voyeur System Structure.

## 5. Conclusions

The Voyeur prototype provides easy construction of new and flexible views for parallel debugging that have greatly eased the laborious task of finding obscure bugs in Poker programs. The current structure has a fairly high degree of flexibility both in the power of the views and in the creation of new views.

Still, there are many areas for improvement; we are now pursuing these as a part of the *Orca* project. We will to exploit the taxonomy of views, possibly using an object hierarchy to classify and create modifications of old views. The setup file used in the Icon and Vector Views is a weak attempt at exploiting class similarities. We need to explore new views. For instance, a new view to log the human-readable messages received from the simulator and provide search and elision capabilities within the messages would help ferret out the worst of the low-level bugs. Increasing the flexibility of the existing views is another goal. For instance in the Icon View, it would be nice to allow the user to select the viewable icons while running the simulator. Also, logical zooming is a powerful tool. For instance zooming out using the Icon View could replace the icons with smaller icons and eventually just a dot for each icon. Finally, we need to explore using Voyeur with other programming systems, such as Presto [15], which supports the development of multi-threaded C++ programs.

## 6. References

- [1] C.L. Seitz. The Cosmic Cube. *Communications of the ACM* 28, 1, pp. 22-33 (January 1985).
- [2] T.W. Pratt. The PISCES 2 Parallel Programming Environment. *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 439-445.
- [3] A.A. Hough and J.E. Cuny. Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation. *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 735-741.

- [4] B.R. Engstrom and P.R. Capello. The SDEF Systolic Programming System. *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 645-652.
- [5] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer* 17,7, pp. 27-36 (July 1984).
- [6] L. Snyder and D. Socha. Poker on the Cosmic Cube: The First Retargettable Parallel Programming Language and Environment. *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 628-635.
- [7] L. Snyder. Introduction to the Configurable Highly Parallel Computer. *IEEE Computer* 15,1, pp. 47-56 (January 1982).
- [8] C.L.Seitz. The Cosmic Cube. *Communications of the ACM* 28,1, pp.22-33 (January 1985).
- [9] A.K. Dewdney. Computer Recreations, *Scientific American*, pp.18-22 (December 1984).
- [10] Dbx. UNIX User's Manual Reference Guide. 4.2 Berkeley Software Distribution. USENIX Association. (March 1984).
- [11] W.P. Crowley, C.P. Hendrickson, T.L. Rudy. The Simple Code. Technical Report UCID-17715, Lawrence Livermore Laboratory (February 1978).
- [12] K. Gates. Personal communication.
- [13] M.H. Brown. *Algorithm Animation*. Ph.D. Dissertation. Technical Report CS-87-05, Department of Computer Science, Brown University (April 1987).
- [14] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics* 5,2, pp. 79-109 (April 1986).
- [15] B. Bershad, E.D. Lazowska, and H.M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Technical Report 87-09-01, Department of Computer Science, University of Washington (September 1987).